HIGH SPEED SIMULATOR - A SIMULATOR FOR ALL SEASONS

K, Patel, W. Reinholtz, & W. Robison

California Institute of Technology
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91109
FAX: (818) *393-1* 178. E-mail: *keyur.c.patel@jpl.nasa.gov*
FAX: (818) 393-1178. E-mail: william.k.reinholtz@jpl.nasa.gov
FAX: (818) 393-1178. E-mail: wilmer.j.robison@jpl.nasa.gov

ABSTRACT: This paper will discuss the evolution of the Multimission Ground Systems Office's (MGSO) High Speed Spacecraft Simulator (HSS) development at the Jet Propulsion Laboratory. This paper will examine the evolution from both a development and operations perspective. The High Speed Spacecraft Simulator (HSS) in reality is a series of simulators capable of modeling the spacecraft and its subsystems at either the bit or functional level, depending on specific mission needs, An initial delivery of the HSS was made to the Galileo (GLL) Project for the sole purpose of validating the GLL Low Gain Antenna mission's flight software resulting from the stuck High Gain Antenna last year. Due to the excellent performance of the HSS in assisting with the flight software validation additional opportunities were identified for its use on the GLL Project. These opportunities included modeling of other on-board data systems, e.g. Command and Data subsystem, Attitude and Articulation Control subsystem. In addition to the data system simulator and the flight software validation capabilities, GLL has replicated the HSS for the purposes of sequence validation and anomaly investigation to name a few. Because the HSS is a software emulator of the flight system, the replication costs associated with adding additional "testbeds" are not only minimal but required no additional maintenance manpower to support. The HSS provides an unparalleled set of capabilities and sophisticated tools for supporting the operational needs of a project while providing detailed visibility into the internal workings of the spacecraft. Today, JPL's MGSO organization is actively developing High Speed Spacecraft Simulators for both the Cassini and Voyager Projects.

The work described in this paper has been carried out by personnel at the Jet Propulsion Laboratory, California Institute of Technology, under contract to the National Aeronautics and Space Administration.

1. INTRODUCTION

As the complexity of spacecraft and missions increase, the verification of commands and sequences by use of conventional means becomes difficult and costly. The construction and verification of commands and sequences has traditionally been a labor intensive and painstaking process. The use of a hardware testbed requires a team of engineers to operate and maintain the equipment and facility. in addition, the hardware testbed only provides a single simulation platform due to the development of additional copies being cost prohibitive.

High fidelity spacecraft simulation involves the execution of the actual binary flight software loads on CPU emulators, interacting with the remainder of the simulator just as it would with flight hardware. As such, both compute-based command errors and the violation of latent constraints can be detected. With the advancement of desktop workstations and their environments it is possible to perform high fidelity simulations at least at real-time and often significantly faster than real-time.

2. I I lSTOR%

Since the advent of microprocessors in spacecraft avionics, it has been considered too complex to perform a rapid simulat ion of the data operations of a spacecraft bit for bit in order 10 test and validate planned events. However, advances in computing technology have provided machines with previously unimagined capabilities which have made this job feasible. In 1991 the High Speed Simulation (HSS) effort was proposed and approved.

Initial protot ypcs [1] of the GLL (which incorporates a data system consisting of six RCA 1802 microprocessors running at 200 KHz and two spacecraft data bum running at 400 KHz) HSS were developed on distributed memory systems interfaced through serial buses. Data transfer rates and latency were critical and eventual 1 y it was found that a shared memory host architecture provided the optimal performance[2,3]. An 8 processor Silicon Graphics computer with 40 MHz R3000 processors and 2 stage cache interfaces to shared memory was used for protot ypes. With this configuration, it was possible to run a simulation 10 times faster than real time and synchronize processes at 75 Hz in simulated spacecraft time (-750 Hz in real time.) This provided the performance needed to build a too] adequat c 10 support spacecraft operations.

MGSO is in the process of delivering production versions of HSS to Cassini (four 1.25 MIP 1750a microprocessors connected by 1MHz 155 3b data buses) and Vo yager projects (very slow custom processors), hosted on a Sun SC2000, with twelve SO-MHz SuperSPARC processors, two-level cache (20Kb instruction, 16Kb Data, 1Mb external), 384Mb of RAM, and 8 GB disk. The Cassini version of HSS, when completed, will run at least seven times faster than real-time, allowing all sequences to be simulated before transmission to the spacecraft. The Voyager HSS, when completed, will run at least 700 times faster than real-linlc.

3. HSS OVERVIEW

The HSS architecture is based upon components (figure 1,). There is a runtime library of components (e.g. CPU models, bus models, other spacecraft hard ware models, simulation schedulers, data viewers, upli nk a nd do wnlink interfaces) each of which has zero or more interfaces (all standardized) and can be connected at runtime to any other model that supports a compatible interface.

This architecture provides a number of advantages. The two most important being strongly-typed runtime component interconnections (called "Splices") and strong conceptual partitioning. The former insures that incorrect interconnections arc rejected. The latter is a consequence of the fact that a component's interaction with the system is fully specified by its interfaces, and thus the semantics of a component can bc understood without having any understanding beyond the i nterfaces. in order to construct a worki ng simulator, a number of simulation components must be created and each of their interfaces spliced to a complementary interface. A simulation component has no internal knowledge of what it is connected to (other than internal assumptions about what an interrupt line means, for example), so it is possible to connect any compatible object to its splices. Onc usc of this is to interpose a monitoring object between two components that would normally be spliced directly together, which can perform extra services such as statistical analysis or graphical display.

The spacecraft systems to bc modeled typically consist of a number of CPUs interfaced via multiple high speed buses and often include several smart peripherals. Such a system is best simulated on a shared-n~cn}ory multi-processor host. The spacecraft data system simulation t ypicall y consists of one process for each significant spacecraft component, all executi ng simultaneously on separate host processors. '1'here arc usually about ten such processes. Each process advances its components for an appropriate time slice, then all processes synchronize to assure that they stay in relative time step. The slice size is usually the largest such that causality is not violated (usually ten to a hundred per second), but in some cases (e.g., debugging) the slice may be a microsecond, or even less.

C++ was chosen as the implementation language for simulation components, because of its run time efficiency and its support for object-oriented programming. A processor component, for example, may emulate a hardware instruction set and the code which implements this must be sufficiently fast to meet the performance objectives of the simulator

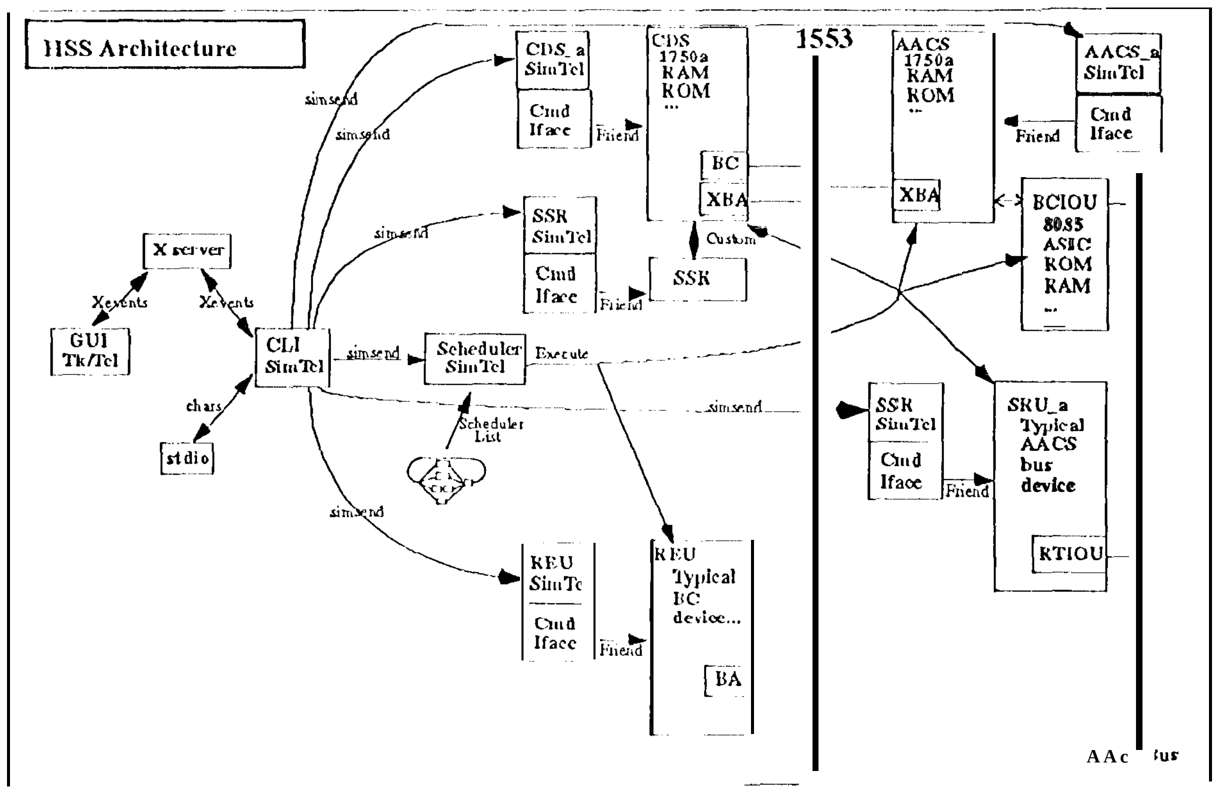

Figure 1: HSS Architecture Diagram

Most components contain' a embedded interpreter which recognizes the language Tcl. Tcl is a freely available embeddable language and concrete interpreter implementation developed at the University of California, Berkeley. Tcl was augmented to make it multi-thread ("MT") safe and to add additional features, in particular a C++ class wrapper and a fast "remote procedure call" that can execute commands remotely on named interpreters (many of the components within HSS cent a in named interpreters, which arc used to manipulate the component) and return the result of the execution to the local interpreter. The HSS 'T"cl subsystem is used for many things, including mode.1 state examination and alterat ion, establishment of model i nterconnections, and model state monitoring and display (using Tk, a toolkit based on 'Tcl which can quickly create Graphical User In(crl'aces). As a general rule, Tcl is u sed unless either performance or robustness concerns dictate the usc of splices.

The current version of HSS includes an Automatic Code Generator (ACG) based on the NASA CLIPS expert system. It is used to generate most of the code that can be deduced from concise specifications. This both makes it unnecessary for the developers to derive the needed code (or even to understand how to do so), and it makes it easy to alter the implementation because though a concept may appear in many places, it only need be altered once in the ACG.

'1'hc 11SS implementation consists of objects which model the large grain hardware components of a spacecraft (e. g., 1750a processors or 1553b bus controllers for Cassini (figure 2.)). It was observed that these components exhibited a natural packaging, with a small number of well defined interfaces. For example, a processor usually contains at least onc read/write interface to an address space. The processor is modeled as an object which i ncludes an cmbcddcd address interface object, and performs reads or writes to addresses via this cmbcddcd object. The address interface object is connected at run lime with another address interface object, which forwards a read or write operation into its owner component for action,

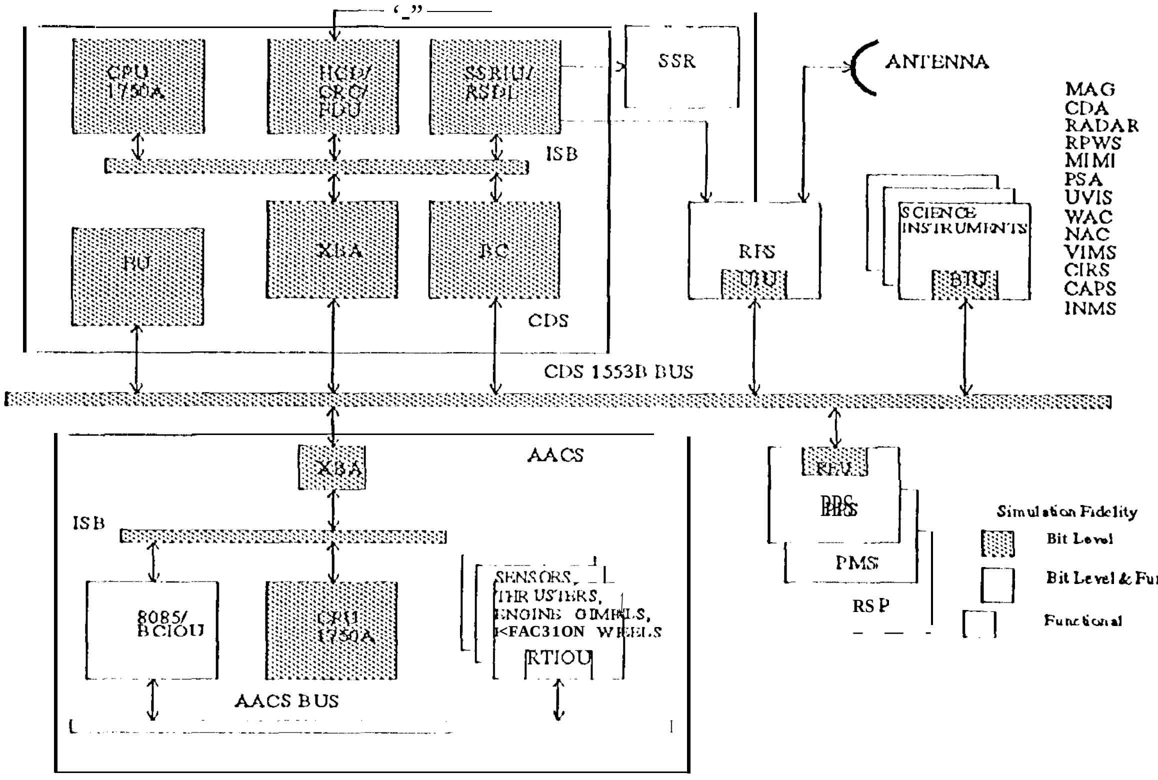

Figure 2: Cassini 11SS A rchitecture

Simulation components usually contain more than onc interface, e.g., processors contain interrupt and other service lines. A simulation component can contain any number of interface objects, including multiple copies of a single kind of interface. A component is used during a simulation run by creating a ncw instance of its type and then hooking up each of its interfaces to a complementary interface.

'1'hc 11SS system includes a bit-level or functional-level simulation of the hardware components of the spacecraft s ystem. Because the components arc decoupled from each other and communicate only via their interface, they can be connected in any way that meets localized objectives for speed and functionality. '1'here can be several component implement aliens for a single processi ng component in the spacecraft, and the choice of which to use can be made at run time. Bit-level simulations arc used where fidelity is of prime concern. I;tlrlctiollal-level simulations arc used where performance is most important.

The design philosophy for simulation components was that they should contain a simple and minimal set of C++ member functions (methods) to implement their functionality, but contain a language interpreter which would allow construction of arbitrarily complicated compound operations based on the atomic member functions. It was felt that it would be hard to anticipate all of the functionality that might be desired from a simulation component, but that by providing a language which could access a component's basic operations we could provide any feature needed without extensive redevelopment.

All components share a small set of common functions, which include an 'execute' command to cause simulation for a specified length of time and a 'set' command' for setting and retrieving the value of a component's internal variables. A component's class definition contains a declaration of all internal variables that will be available to "Tcl via the 'set' command, which can include. processor registers, memories, and other metadata which a component contains such as lists of bus transactions, etc.. Objects which inherit from a base class object receive the same 'Tcl functionality without having to redeclare it. Individual component classes may also extend their command set to provide functions particular to their operation

The simulator operates in an interpreted fashion in constructing a spacecraft model to perform a particular simulation. A single object is initially constructed, the "executive", which is responsible for interpreting further commands to create objects, splice them together, and execute simulation activities. It then waits in a service loop for further commands from a user interface component, if one has been created, or from the standard input if one has not.

The interpreted nature of system construction allows for a great deal of flexibility. It is possible to create a standard simulation from a batch file, or to create a graphical user interface (GUI) which offers choices for configuration options to allow a user to bring up a customized simulator.

Commands to the executive are 'Tcl commands which are special to its class: 'simnew' for invoking component constructors, 'splice' for connecting component embedded interfaces together, etc. Another command which is special to the executive is 'simsend', which allows a "Tcl command to be sent to any object that has been created. This allows script driven ad hoc queries and computation to be performed at any time, giving a large amount of adaptability and flexibility to the simulation system

The simulation can be operated by sending 'Tcl 'execute' commands via the executive's 'simsend' commands, but this is not fast enough for project specific simulation runs, Instead, a ' scheduler' component is created which can form execution splices to simulation components, which add minimal overhead in invoking component execution member functions.

The scheduler is responsible for enforcing rendezvous points during execution, During parallel execution of a simulation, a number of machine cycles for different components may be executed simultaneously and asynchronously, and if any interaction between components occurs at this time it will most likely not reflect the event ordering that occurs on the real hardware. Rendezvous provide synchronization and a merging of threads so that component interactions are properly ordered. Rendezvous times can be set to any value, from as little as one simulation clock cycle to any arbitrarily higher value. A one clock cycle rendezvous ensures perfect fidelity to the spacecraft hardware, but the overhead from this number of rendezvous greatly reduces performance. Setting larger rendezvous values allows us to achieve higher performance, but there is an upper limit to the size of a rendezvous before the simulation fails to mirror the actual hard ware. in the GLL spacecraft, most transactions between spacecraft components occur at a "real time interrupt" (RTI) which occurs every 1/15 second, but a number of sub-RTI transactions also occur. We have found that rendezvousing at 1/5 of an RTI gives the largest time slice that will work, but which still allows us to attain a ten times real time simulation,

Rendezvous points are enforced by the scheduler even when execution commands are sent which would otherwise cause a rendezvous point to be overrun. For example, a user might choose to single step the simulator through an interesting portion, then ask it to jump ahead one RTI. The scheduler issues commands 10 components 10 execute for a particular number of clock cycles, and reads the return value of the call to find the actual number of cycles executed (a component may execute less than the number of requested cycles if it cannot finish an atomic operation). It then computes the correct number of cycles to send to each component on the next command based on the running total of executed time for each component and the length of time to the next rendezvous.

A simulation may be run on a single processor or multiple processor machine with no change in code or configuration. On a single processor machine, a parallel simulation simply executes sequentially.

'Monitors' arc simulation components which arc not part of a spacecraft model, and which serve as companion components that watch and report on the state of a spacecraft component, Like all components, they contain a Tcl interpreter which can provide any desired functionalist y by loading the proper "l'cl program. Monitors can bc created on the fly, and can bc added to or deleted from the scheduler as necessary. Monitors often usc "simsend" to communicate with their companion component, but in some Cases (e.g. examination of a large number of memor y locations) a splice is used, instead

Onc usage of monitors is for debugging: the monitor is set to watch for any interesting state specified by the logic in its script. On detection of this state, it can send a message or a request to halt of the simulation run. It can also bc used to send periodic values back 10 the executive which can bc forwarded to a user interface for display in any desired way, e.g. strip chart, dial, text, etc.

4. OPERATIONS

11SS is dc.signed to fit into the current mission operat ions processes. The simulator lakes i nput files and flight software loads "as is". The 11SS accepts uplink commands in the same way as the actual spacecraft. The telemetry data stream generated by 11SS can bc connected 10 the real ground telemetry handling system of the Ground 1)ata System.

During execution, 11SS generates a file which contains a timestamped list of commands executed by the spacecraft. The tile can be compare.d with a prediction file generated by the MGSO sequence generation software, to validate that commands will bc executed at the correct time on the spacecraft.

The simulation may bc checkpointed at requested intervals. The checkpointed state can be restored at a later lime to continue execution. This function can also bc used after the detection of an anomalous Condition, by restoring the most recentl y checkpointed stale vector and then si ngle step forward unt i] the anomaly is again encou nt crcd. This will allow monitors to do sampling at longer intervals while still providing for a capability to pinpoint an exact anomaly state, without significant y degrading simul ator performance. Multiple copies of the checkpointed state can bc run on multiple machines to explore multiple state space paths concurrently.

11SS provides the capability (called timejump) to rapidly advance time across quiescent periods, thus significantly increasing simulation speed. Because all components of the spacecraft arc available to the simulator, the user can tell the spacecraft that an arbitrary amount of time has passed and thus advance the state of the spacecraft to the new time. The user must be careful that no impel-tant activities arc passed over.

11SS provides the capability to simulate any spacecraft faults, because the complete state of the spacecraft is available during the simulation, The user must identify the signature of the fault.

The simulator can inject that signature onto the spacecraft simulation. The user can then see how the spacecraft reacts to the simulated fault

11SS's unsurpassed visibilit y plus it's rcxd-time environment provides an excellent environment to test flight software. Flight software developers have been hampered by debuggers that are not real-time oriented, They are sufficient for testing paths within routines, but are not sufficient to validate the correct operation of flight software at the spacecraft level. Traditionally, hardware test platform are in short supply and generall y have to be shared by software and hardware developers. 11SS can provide any number of test platforms with much more visibility than is generall y avai lable from hardware lest plat forms.

Much of 11SS's visibility is derived from its use of the. 'J'cl scripting language. Each component of the spacecraft being simulated is designed with visibility in mind so that the user may view and modify registers, memory, etc. which are of interest, Because the visibility is accessible from 'I'cl, the user may generate condition checking of arbitrary complexit y. The user can test for the combination of any number of variables with full arithmetic and Boolean functions. The 'J'cl scripting language also allows the user to automate the execution of the simulator. The user may compose a script commanding the simulator appropriatel y. The user at any time may validate that any conditions expected are still valid. The 11SS developers use this capability to run regression tests nightly.

5. FUTURE DIRECTION

A number of ways have been identified to reduce the cost of developing HSS simulators. The theme common to the ideas outlined below is to move the representation of the simulation more towards the application domain, and away from the development domain. This will reduce the effort required to create simulators, because less translat i on from speci fication to implement ation will be required, and because we will be able to t ake bet ter advant age of domain expertise. In other words, one day those that underst and the spacecraft archi tect ure will be directl y involved in the creation of the simulation, because esoteric development knowledge and skill will not be required.

GU1-based simulation construction - '1'here is no convenient and "intuitive" way of representi ng the design of a spacecraft simulation. It would not be too difficult to build a GU1-based spacecraft builder that allows the simulation designer to select components and specify their interconnections. This should require less tool skills of the designer than the current script-based notation, and so allow individuals with domain knowledge, but not profound tool knowledge, to do such work.

Gene.rate models directly from specifications - There are often hardware-orien[cd (e.g. VHDL or Verilog) model specifications available, created as a byproduct of building the spacecraft hardware. Conventional simulation of such products is far too slow, but there is promising work suggesting that it may become possible to execute such models at the functional level with relaxed fidelity, with performance sufficient for our needs. This would reduce the 11SS development cycle time and cost, and i ncrease reli abilit y, because the currentl y human-intensive process of converti ng hardware specifications into high-performance 11SS models might one day be automated.

User-programmable models - Most models, especiall y complicated ones, are writ ten in C++ and so to work on such a model, the developer must be a programmer as well. Consideration has been given to implementing models that directly execute domain-oriented notations (e.g., s(atc-based, ru]c-based, or procedural) so that (1) the model implement at ion is more c1osc1 y rel ated to the specificat ion; and (2) domain experts can write or inspect the models more easily.

6. CONCLUSION

11SS has shown that high speed simulation is possible for complex spacecraft systems with high fidelit y. The 11SS can be used for a variety of testing ranging from sequence to flight software. The HSS has been successfully used on GLL for both sequence and flight software testing. Later this year HSS will be delivered to both Cassini and Voyager projects.

7. REFERENCES

1) John E. Zipse and Raymond Y. Yeung, "A Multi-Mission High Speed Spacecraft Simulator Concept", l'reposal June 1991.

2) Alan Morrissett et al, "Multimission High Speed Spacecraft Simulation for the Galileo and Casani Missions", AIAA Computing in Aerospace 9, 1993.

3) W. K. Reinholtz and W. J. Robison, "The ZIPSIM Series of High Performance, High Fidelity Spacecraft Simulators", 8th Annual AIAA/USU Conference on Small Satellites, August 1993.